IEEE Transactions on Evolutionary Computation 13.3 (2009): 477 – 485

DOI: http://dx.doi.org/10.1109/TEVC.2008.2008797

Towards the Validation of Plagiarism Detection Tools by means of Grammar Evolution

Manuel Cebrián, Manuel Alfonseca, *Member, IEEE*, and Alfonso Ortega

***Abstract*— Student plagiarism is a major problem in universities worldwide. In this letter, we focus on plagiarism in answers to computer programming assignments, where students mix and/or modify one or more original solutions to obtain counterfeits. Although several software tools have been developed to help the tedious and time consuming task of detecting plagiarism, little has been done to assess their quality, because determining the real authorship of the whole submission corpus is practically impossible for markers. In this letter we present a Grammar Evolution technique which generates benchmarks for testing plagiarism detection tools. Given a programming language, our technique generates a set of original solutions to an assignment, together with a set of plagiarisms of the former set which mimic the basic plagiarism techniques performed by students. The authorship of the submission corpus is predefined by the user, providing a base for the assessment and further comparison of copy-catching tools. We give empirical evidence of the suitability of our approach by studying the behavior of one advanced plagiarism detection tool (AC) on four benchmarks coded in APL2, generated with our technique.**

***Index Terms*— Plagiarism detection, Computer Programming Assignment, Benchmark, Grammar Evolution.**

I. Introduction

Undergraduate student plagiarism is becoming one of the biggest problems faced today by universities worldwide [4]. Two main types of documents are targets of plagiarism: essays and computer assignments, although cases in art degrees have also been reported [23, p. 4]. In this letter we focus on computer assignments.

Every computer science lecturer knows that plagiarism detection (copy-catch) is tedious, and extremely time consuming. Several plagiarism detection tools have been implemented since the 1960s: MOSS [2], SIM [11], YAP [13], JPlag [19], SID [6] and recently the integrative AC [9], to name the most widespread in the academic community.

The problem we are interested in occurs when facing the assessment of such tools. Quoting Whale [22, p. 145]: "Assessing different techniques for similarity detection is possible only on a relative scale". Although the work by Whale dates from seventeen years ago, the essence of its consideration remains up to to date. The reason is very simple: it is almost impossible to determine whether an assignment solution is a plagiarism of another. What is more, in some cultures (as for example the one in which the authors have an extensive lecturing experience), a student will deny a plagiarism even in the most blatant cases. Of course this is not the case in other cultures, where students who have admitted plagiarism can be found.

In the former case, the decision of whether a solution is original or not is a matter of judgment and generally depends on the sensitivity of the marker to find abnormally similar works. This subjectivity may contaminate benchmarks constructed in this way, thus little accuracy may be expected in the assessment.

M. Cebrián is with the Department of Computer Science at Brown University, Box 1910, Providence, RI 02912, USA (e-mail: mcebrian@cs.brown.edu).

M. Alfonseca and A. Ortega are with the Escuela Politécnica Superior at Universidad Autónoma de Madrid, P. O. Box 28049, Madrid, Spain (e-mail: {manuel.alfonseca, alfonso.ortega}@uam.es).

Acknowledgement: This work has been sponsored by the Spanish Ministry of Science and Technology (MCYT), project number TIC2002-01948.

It is interesting to notice that there are plenty of examples of actual plagiarized code from students which can be used as benchmarks (see e.g. [15]). Unfortunately most of them are not in the public domain.

Two main attempts to ameliorate this issue have been carried on. The first [10] consists of performing edit operations on a solution to obtain a plagiarized one: variable and function name renaming, comment removal, inversion of adjacent statements, permutation of functions, etc. The problem with this approach is the high perceptiveness and time needed to perform this task, generally resulting in benchmarks of small size, as they have to be created by hand. The second (less ambitious) attempt [6] builds plagiarized assignment solutions by means of the random insertion of irrelevant statements into the original code, in the hope of confusing the detection mechanism.

We feel that a more principled approach is necessary in order to perform a fair comparison of detection tools. In this letter we present a technique which, fed with some realistic specifications and the grammar of a programming language, is able to generate benchmarks of the desired size. Each benchmark is made of a subset containing independent solutions to the specifications, coded *from scratch*, and another subset - the plagiarized solutions - built from one or two solutions taken from the original subset. Both the authentic and the plagiarized sets are built by means of evolutionary techniques adapted from Grammatical Evolution [17], whose suitability for automatic programming is well established.

In this letter we try to show that having an arbitrary number of large solutions to an assignment, with a priori knowledge of their phylogeny, is the first step towards a benchmark for plagiarism.

The remainder of the letter is organized as follows: in sect. II we detail the benchmark generation technique; in sect. III we give experimental evidence of the suitability of this technique through several examples. Sect. IV discusses the usefulness of our approach for the generation of benchmarks. Sect. V proposes some conclusions and possibilities for improvement.

II. Automatic generation of benchmarks

Our benchmarks simulate the answers of different students to a practical assignment. In this letter, each benchmark consists of APL2 functions which fit a set of points generated by applying one particular function to the set of inputs (values of x) $1, 2, 3, 4, 5$. Four benchmarks have been generated, corresponding to the following toy problem functions: $x^2, 1 + x + x^2 + x^3, \cos(\log x)$ and $\log(x^3)$.

As a first step towards mimicking the solutions of the students to this assignment, two sets of programs are generated for each benchmark: the first is considered *original*, the second contains plagiarisms. Both sets are built by means of a genetic engine in two phases: in the first, 30 original programs are generated using a grammar evolution inspired technique (GE) [17]. Then 14 solutions are generated by applying several selected genetic operators, which produce small changes in the solution's genotype and lead to usually minor modifications in the corresponding source code. These figures (30 and 14) have been chosen to correspond, approximately, to the maximum number of students (44) in the programing laboratories of the institutions in which the authors lecture and in other institutions visited. The number of plagiarisms has been slightly overrated to simplify the use of the benchmark.

All the solutions consist of an APL2 function with the same header: the name of the function is F, their input is argument X, and their return value is variable Z. The first instruction assigns the value of X to Z to guarantee that F always returns some value. In the original solutions, F contains a number of additional instructions, between

```
E::= O | oO  | OoO
O::= 0|1|2|3|4|5|6|7|8|9|
     X|X|X|Z|Z|Z|Z|Z|Z|Z|
     (E)|(E)|(E)|(E)|(E)|(E)|(E)|(E)|(E)|(E)
o::= +|-|×|÷|⌊|⌈|*|○|⊛|!
```

Fig. 1. Context free grammar to generate and modify the original APL2 functions. The repetition of a symbol affects the probability of its choice.

0 and 255. Each one assigns the value of an expression to variable Z. These expressions are generated by means of GE. Fig. 1 shows the context free grammar used to generate the expressions. E is the axiom. A genotype consists of a number (between 100 and 200) of integers (codons) in the [0,255] interval. The first codon indicates the number of instructions to be added to the function. The genotype is mapped in the usual GE way and derives the number of expressions indicated by the first codon from the initial word E. The alternate execution mechanism provided by APL2 has been used to intercept semantic errors in the generated expressions, thus avoiding program failures and unexpected end conditions. Each instruction is executed in the same way and occupies a single line, therefore the size of the generated APL2 function is equal to the value of the first codon plus one (for the header).

The fitness function is the mean quadratic error of the generated APL2 function applied to the set of control points, as compared with the set of control results, scaled by a factor to punish long genotypes (size(genotype)/100), to favor *parsimonious* answers. The fitness optimal value is 0. The experiment stops when the solution found has a fitness value less than 1 or when the number of generations equals 1000. The genetic operators used are taken from *mutation with elision, mutation with elongation, genotypic recombination* and *phenotypic recombination.* [17].

In the generation of the 30 original solutions we have used 30 different mono-individual populations with one independently generated genotype each (corresponding to 30 different random seeds), equivalent to performing a hill-climbing local search. The genotype of the next population is obtained by applying mutation with elision to the previous individual, which is either mutated or shortened with the same probability (0.5). Elision deletes a codon in an arbitrary location of the genotype. The new genotype replaces the old one only if its fitness is better.

Mutation with elongation is similar to mutation with elision: an arbitrary codon is added in a random location of the genotype, rather than being deleted. Each time the operator is applied on the genotype, the process is repeated 5 times. This genetic operator tries to simulate a student performing random changes (adding and replacing a few fragments) to an original source in the hope of differentiating the plagiated code from the original one, to avoid being caught. The changes performed are random and will usually worsen the correctness/fitness of the program, as happens in real life.

One point recombination is used in genotypic and phenotypic recombination. In our approach, only the child that begins like its first parent is taken into account. If we want to obtain two children, the same parents may be used in the opposite order, although in the second case the recombination point will usually be different. The procedure is performed 5 times, and the child with the best fitness is selected as the result of the recombination. This genetic operator is intended to mimic the typical behavior of a student who possesses two original solutions. The student understands both solutions to some degree and tries to mix them in several (5) ways, retaining the most successful one. A good understanding of the assignment is assumed, and is reflected in the fact that the mix is done at the genotypical level, in contrast with a simple cut&paste.

Phenotypic recombination acts directly on the APL2 functions, so each child will contain the first lines of one parent and the remaining instructions of the other parent. This operation is complementary to the one modeled in the previous paragraph: here the student would have a superficial understanding of the assignment and the two solutions, and performs a simple cut&paste operation to obtain the plagiarism.

Although copies from a single source are much more frequent than from multiple sources, we have included this option because we have found empirical evidence of its presence during the 2-year extensive use of the anti-plagiarism tool AC [9] in a real academic environment.

We have applied these techniques to plagiarize one or two original functions. The 5th, 10th, 15th, 20th, 25th and 30th original solutions are plagiarized using mutation with elongation or elision, to generate 6 new solutions. Next 4 new APL2 functions are generated through the genotypic recombination of the following pairs of originals: 5th and 10th, 10th and 5th, 15th and 20th and 20th and 15th. Finally phenotypic recombination is used to mix the 20-15, 7-14, 5-22, and 30-1 pairs. Fig. 2 shows a graphic scheme of the whole process and fig. 3 shows the existing plagiarism relations in the benchmarks.

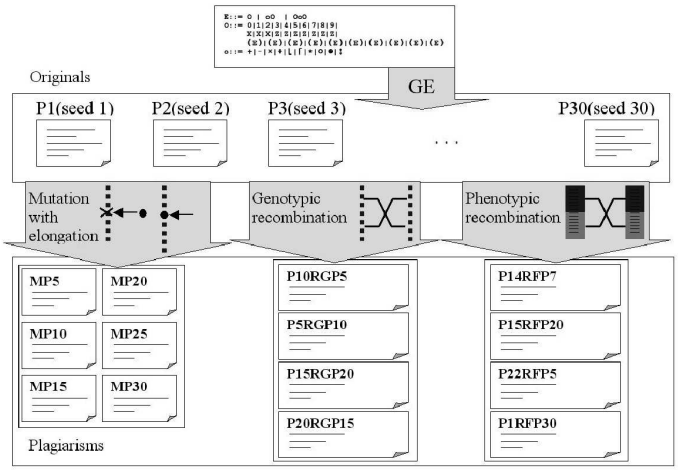

Fig. 2. Graphical scheme of the whole process

The APL2 choice

The APL2 language has been selected as the language in which the benchmarks are coded for the following reasons:

- APL2 is a very powerful language, especially for the generation of expressions, with a large number of primitive functions and operators available.
- The APL2 expression grammar is very simple and can be implemented with just three non-terminal symbols, which simplifies the grammatical evolution process.
- APL2 instructions can be protected to prevent semantic and execution errors giving rise to program failures. In this way, we can rest assured that all the programs in the benchmark will execute (although their results may not be a good answer to the assignment). The Grammar Evolution technique is also simplified, because we do not need to include any semantic information, as in attribute grammars or Chistiansen's grammars [8], [18].
- APL2 makes it possible to define new programming functions at execution time, thus providing the feasibility of integrating

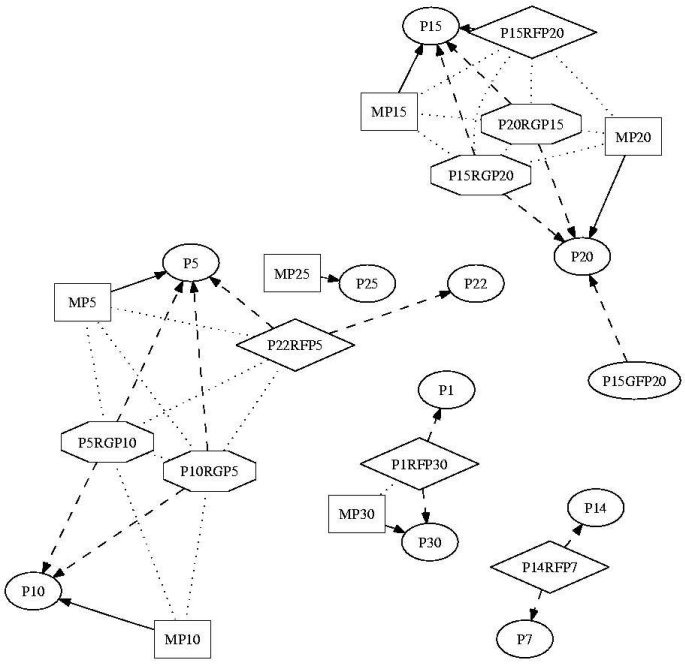

Fig. 3. Plagiarism relations of the benchmarks. Round vertices stand for original submissions, squares for plagiarism using a single source, rhomboids and octagons for the two different types of plagiarism using two sources. A solid line between vertices A and B denotes that A has used B as the unique source of plagiarism; a dashed line between A and B denotes that A has used B as one of the two sources of plagiarism; a dotted line denotes indirect copies, i.e. those which share a common source of plagiarism.

the fitness computation with the genetic algorithms generated by the benchmark. With a compilable language such as C, this would be very difficult. For a short introduction to the APL2 language see [3].

III. Experimental results

Summarizing: we have generated 4 benchmarks, each consisting of 44 submissions coded in APL2. Each benchmark is divided in the same manner:

- 30 original solutions, named P1 to P30.
- 6 *mutational plagiarized results*, named MPx, where x stands for the original source of plagiarism (5, 10, 15, 20, 25 and 30).
- 4 *genotypic recombination plagiarized results*, named PxRGPy, where x and y represent the two source genotypes used as parents in the genotypic recombination; y is considered to be the first parent.
- 4 *phenotypic recombination plagiarized results*, named PxRFPy, where x and y represent the two source genotypes used as parents in the phenotypic recombination; y is considered to be the first parent.

As indicated in the previous section, the specifications of the 4 benchmarks were the functions x^2, x^3+x^2+x+1, $\cos(\log x)$ and $\log x^3$. Some statistics of the generation process are shown in table I. Executions took about one hour per benchmark on a 2.5 GHz computer with 512 MBytes memory.

We used the plagiarism detection tool AC [9] to check whether the sets generated with this process capture some basic elements found in real plagiarisms. To do this, we fed our 4 benchmarks into AC, which works in two steps: first, one of the similarity metrics available is selected[1] by the end-user of the tool (the marker). Then, after

[1]Ranging between 0 (complete similarity) to 1 (complete dissimilarity).

	ave. program size	ave. instructions
x^2	1889	120
x^3+x^2+x+1	1954	126
$\cos(\log x)$	2349	140
$\log x^3$	1735	108

TABLE I
STATISTICS OF THE GENERATION OF THE FOUR BENCHMARKS (THE AVERAGE PROGRAM SIZE IS MEASURED IN BYTES).

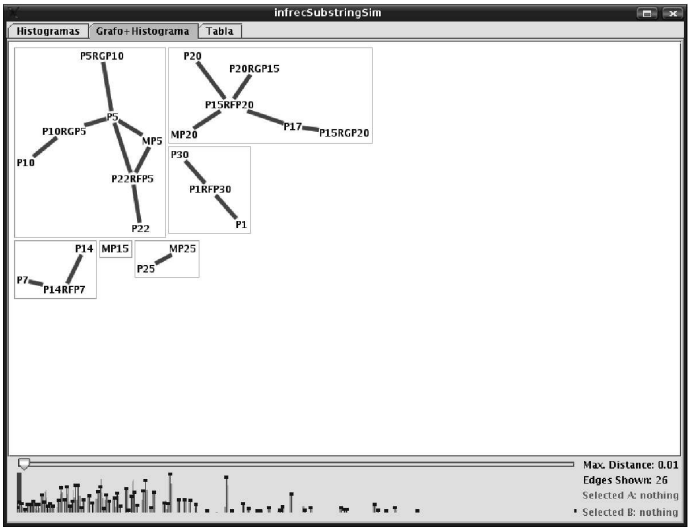

Fig. 4. The vertices of the graph stand for each submission of the benchmark for x^2, while the edges represent values of pairwise distances calculated using the longest most infrequent similarity distance. Only the submissions whose pairwise distance is lower that the distance chosen by the slider (below) are shown. In this figure, the slider is set to 0.01.

the pairwise distances between all submissions are obtained, several graphical interfaces are displayed to point out abnormal low distances which could imply a plagiarism.

Fig. 4 displays a similarity graph obtained by computing another similarity distance on the benchmark x^2. This distance finds the longest-most infrequent string which two submissions have in common; the longer and the more infrequent the string, the lower the distance between the solutions. A graph is provided by the tool, whose vertices stand for each submission solution and whose edges represent the distances between each pair of solutions. Only distances smaller than the value chosen with the slider are shown. This graph constructs and displays minimum spanning trees (MSTs) built only with those distances below the threshold, 0.01 in this figure. It can be seen that the obtained MSTs are exactly what one would desire: plagiarized versions clustered with their sources, in all cases but submission P17, which is a typical case of an accidental coincidence. In fig 5, where the threshold has been increased to 0.02, the overwhelming majority of the plagiarized versions have been detected (13 out of 14), against only one additional non-plagiarized MST (P3-P28), i.e. plagiarized versions tend to appear long before non-plagiarized ones.

Fig. 6 shows the results for a different benchmark, function $\cos(\log x)$. The measure of distance used is the normalized compression distance (NCD, see [7]) which, in simple terms, gives a low distance to sources which compress well together, i.e. which share a large amount of literal coincidence. Finally, the visualization is based on *individual hue histograms*, meaning that the darker the color, the more elements are in this range. Each row displays the histogram of

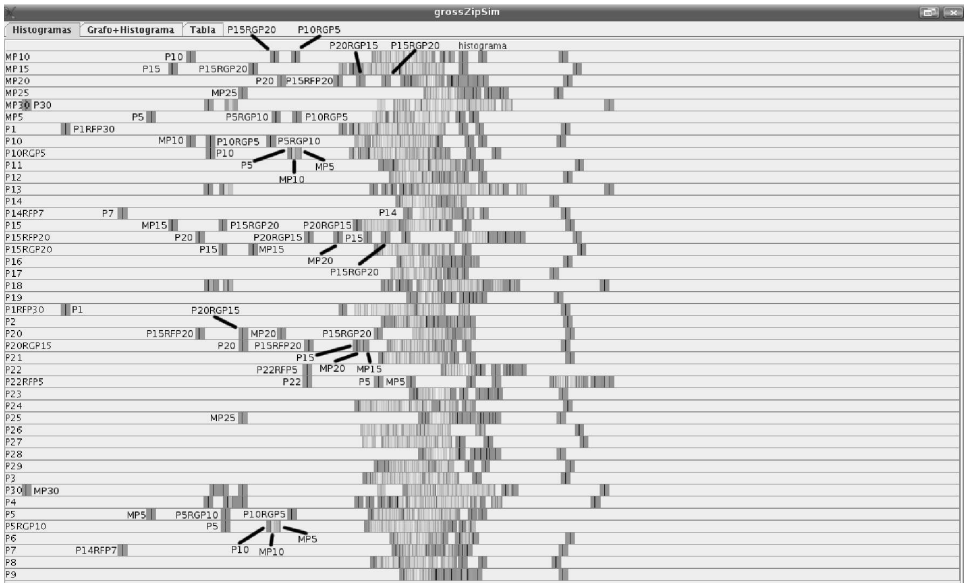

Fig. 6. We explain the first row, the others are similar. The pairwise distances are computed between MP10 (leftmost part of the row) and the rest of submissions for the $\cos(\log x)$ corpora. We then depict a histogram of the distances, where a darker color at a certain distance represents a higher number of submissions lying at that distance from MP10. The horizontal axis of the histogram ranges from 0 (leftmost, complete similarity) to 1 (rightmost, complete dissimilarity).

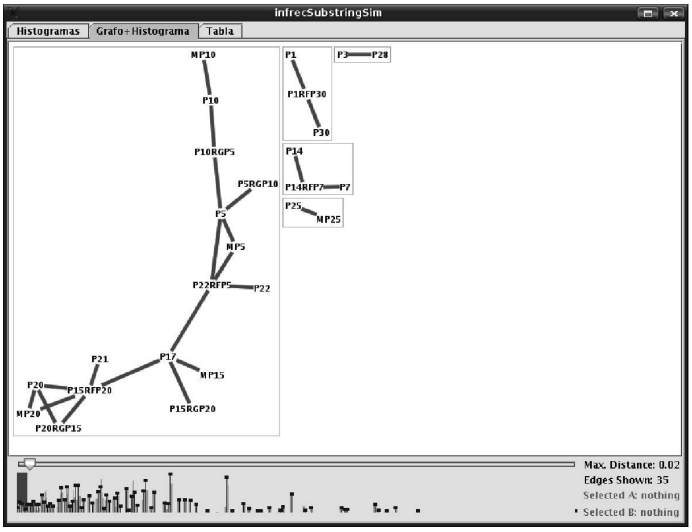

Fig. 5. Analog to fig. 4, but with threshold increased (slider set to 0.02).

x^3+x^2+x+1		
9.881421E-4	P10	MP10
0.0014822131	P15RFP20	P15
0.0014822131	P20RGP15	P15
0.0014822131	P20RGP15	P15RFP20
0.0019762842	P30	P1RFP30
0.0019762842	MP20	P20RGP15
0.0024703552	P15RGP20	P20RGP15
0.0029644263	MP30	P1RFP30
0.0029644263	MP30	P30
0.0039525684	P10RGP5	MP10
0.004446639	P18	P7
0.004446639	P18	P14RFP7
0.0049407105	P26	P4
0.0049407105	P26	P12
0.0049407105	P26	P18

TABLE II

LOWEST 15 PAIRWISE DISTANCES OBTAINED USING THE LONGEST-MOST INFREQUENT DISTANCE ON THE BENCHMARK x^3+x^2+x+1.

NCD distances between the submission in the leftmost part of the row and the rest of the benchmark. It can be seen that plagiarized versions are nearer to their sources than to any other, at distances usually *outlying* from the rest of the sample.

Another option available in AC provides a raw list of pairs sorted by their increasing chosen distance. In tables II and III we display the 15 lowest distances for benchmarks $\log x^3$ and x^3+x^2+x+1, where the NCD and the longest-most infrequent distances are used respectively. In both, plagiarized sources or pairs of sources plagiarized from the same source are generally top ranked, specially in the case of $\log x^3$, where no non-plagiarized pair appears in the table. Therefore, even if no graphical help is used, plagiarized pairs manifest by themselves.

IV. RELATING PLAGIARISM TO FUNCTION OPTIMIZATION

In sections II and III we have tried, first conceptually and then empirically, to show that copies generated by our procedure capture a basic element found in plagiarisms: an improbably high similarity between works submitted by different authors. If we consider this definition in depth, we find that a philosophical problem shows up:

Assume that students have some specifications for an assignment and there exists only one optimal way to code the solution. We consider as optimal the following:

- Perfect functionality: for every input, the computer program must produce the specified output.
- Maximal parsimony: the program must be as simple as possible. During the generation process, solutions with a high number of lines are penalized, although other measures of parsimony can be used [20], [14], [5], [21].

It is possible that a single solution exists with perfect functionality and maximal parsimony. These conditions are not very restrictive if,

$\log x^3$		
0.01538462	P1	P1RFP30
0.02339181	P15RFP20	P15
0.02339181	MP15	P15
0.02339181	MP15	P15RFP20
0.02469136	MP25	P25
0.13580246	P25	P20RGP15
0.13580246	MP25	P20RGP15
0.15789473	P20RGP15	P15
0.15789473	P20RGP15	P15RFP20
0.16374269	MP15	P25
0.16428572	P10RGP5	P5
0.16959064	P25	P15
0.16959064	P25	P15RFP20
0.16959064	MP15	P20RGP15
0.16959064	MP25	P15

TABLE III
LOWEST 15 PAIRWISE DISTANCES OBTAINED USING NCD ON THE BENCHMARK $\log x^3$.

```
Z←F X
Z←X
'' ⍎EA 'Z←7↓7' [trash]
'' ⍎EA 'Z←-Z'  [trash]
'' ⍎EA 'Z←((×7)+(6))×Z' [trash, shared]
'' ⍎EA 'Z←-Z' [trash, shared]
'' ⍎EA 'Z←⌈(⌈X)' [trash, shared]
'' ⍎EA 'Z←2×Z' [trash, shared]
'' ⍎EA 'Z←÷X'  [trash, shared]
'' ⍎EA 'Z←(Z↓4)' [trash, shared]
'' ⍎EA 'Z←5' [trash, shared]
'' ⍎EA 'Z←(⌊((Z)-(↓X)))×8' [trash, shared]
'' ⍎EA 'Z←3○(⌊Z)' [trash, shared]
'' ⍎EA 'Z←×6' [trash, shared]
'' ⍎EA 'Z←X' [trash, shared]
...
```

```
Z←F X
Z←X
'' ⍎EA 'Z←7↓9' [trash]
'' ⍎EA 'Z←9'   [trash]
'' ⍎EA 'Z←Z'   [trash]
'' ⍎EA 'Z←((×7)+(6))×Z'[trash, shared]
'' ⍎EA 'Z←-Z' [useless, shared]
'' ⍎EA 'Z←⌈(⌈X)' [trash, shared]
'' ⍎EA 'Z←2×Z' [trash, shared]
'' ⍎EA 'Z←÷X'  [trash, shared]
'' ⍎EA 'Z←(Z↓4)' [trash, shared]
'' ⍎EA 'Z←5'  [trash, shared]
'' ⍎EA 'Z←×(((Z)-(↓X))×8)' [trash, shared]
'' ⍎EA 'Z←3○(⌊Z)' [trash, shared]
'' ⍎EA 'Z←×6' [trash, shared]
'' ⍎EA 'Z←X' [trash, shared]
...
```

Fig. 7. Two fragments of code of P15 (above) and MP15 (below) from the $\cos(\log x)$ benchmark. Dots "..." stand for code not shown.

```
...
'' ⍎EA 'Z←Z○Z' [trash, shared]
'' ⍎EA 'Z←X'   [trash, shared]
'' ⍎EA 'Z←0×Z' [trash, shared]
'' ⍎EA 'Z←Z'   [trash, shared]
'' ⍎EA 'Z←-Z'  [trash, shared]
'' ⍎EA 'Z←ėX'  [trash]
'' ⍎EA 'Z←×X'  [trash]
'' ⍎EA 'Z←Z'   [trash]
'' ⍎EA 'Z←Z'   [trash]
...
```

```
...
'' ⍎EA 'Z←Z○Z' [trash, shared]
'' ⍎EA 'Z←X'   [trash, shared]
'' ⍎EA 'Z←0×Z' [trash, shared]
'' ⍎EA 'Z←Z'   [trash, shared]
'' ⍎EA 'Z←-Z'  [trash, shared]
'' ⍎EA 'Z←Z'         [trash]
'' ⍎EA 'Z←-((Zė9))' [trash]
'' ⍎EA 'Z←ZėX'       [trash]
'' ⍎EA 'Z←((Z))'     [trash]
...
```

Fig. 8. Two fragments of code of P10RGP5 (above) and P5 (below) from the $\log x^3$ benchmark.

for example, we consider the way in which programming challenges are usually qualified (see [1]).

In this situation, two students delivering the optimal solution to the marker could meet the already mentioned definition of plagiarism: absolute coincidence. The marker would argue that it is highly improbable that two students end up with the same code and consider them plagiarisms, but the students can reject this argument with the easy explanation that they have optimized the program independently until no further improvement was possible. If the programmers are good enough, the probability of reaching the same optimal or quasi-optimal solution is high.

The solution to this problem is provided by the experience of the marker at copy-catching: plagiarism is usually detected rather by observing abnormal coincidences in *trash code*, i.e. erroneous or spurious code, than by coincidences such as similar variable or function names in correct portions of the code. The underlying idea is that there are few ways of doing things correctly, but many of doing them inaccurately, so why would two students chose the same way of making mistakes? Reported cases of copy-catching describe shared lines of code that simply do nothing, or two compiled codes which produce the same errors when executed. This may happen because plagiarists have a poor understanding of the code and often tend to incorporate *trash code* from the source into their code. Even those most daring who try to change some fragments of the code usually fail to notice this, usually worsening that code.

To simulate the plagiarism process, one has to take this into account. It turns out that there is a strong correspondence of these ideas with those of search and optimization: perfect solutions are equivalent to *global optima*, while approximate solutions, those which include trash code, are equivalent to *local optima*.

Our proposed generation process can be seen in this light. First we generate the original solutions, which are desired to be different. To do this, we perform a *light* optimization, i.e., we try to maximize functionality and parsimony, without seeking the global optimum. This is done by limiting the number of optimization steps. What is obtained is a set of local optima.

In a second step, the counterfeits are created. Using genotypical mutation with elongation, a new solution is created which will share a high percentage of code with the original. The shared code will consist of both useful and trash code. On the other hand, the new code generated by the mutation/elongation will probably worsen the fitness of the submission. These new solutions will also be local optima, but hopefully near enough (using some natural similarity distance) to the previous set, having been generated randomly.

Fig. 7 shows code fragments of submissions P5 and MP5 from benchmark $\cos(\log x)$. Shared code and trash code are annotated at the right. Detection is possible precisely because of the shared trash code rather than the useful code, because the latter will be the same in both cases with high probability. The same happens if we consider genotypical (fig. 8) or phenotypical (figure not shown) recombination. The generated codes are mixtures of the sources, where some trash code has been inherited from both. As can be seen in these examples, the trash code should be the fingerprint for plagiarism detection.

In this section we have somewhat focused on functional languages. When procedural languages are considered, trash code may be represented mainly by comments or identifier selection, but even so it would be useful to detect plagiarism, namely by similarity measures that do not tokenize the source code (such as the longest most infrequent string mentioned above).

V. Conclusions and future work

Copy-catching computer tools are difficult to evaluate, because actual work by real students is always subject to uncertainty. To help in their evaluation for the field of computer programming assignment plagiarism, we offer a procedure which automatically generates different benchmarks which may be useful for this purpose. A benchmark for a given assignment is made of a number of original solutions, together with another set of plagiarized solutions, generated in such a way as to capture some basic elements observed in real plagiarisms. We have used these benchmarks to assess the performance of the advanced detection tool (AC), with preliminary satisfactory results.

Despite this initial success, we are aware that APL2 belongs to the family of functional languages, which is not the most widely used in worldwide education nowadays. In the next step we want to extend this work to include classic procedural languages, maybe initially simplified, such as ASPLE [16]. However, being able to feed our method with real world languages such as C or Java is an important cornerstone in this research, as the two *market leader* systems available (JPlag and MOSS) work with those languages and, for example, do not work with APL2 or ASPLE.

This comparison could be done by making some statistical analysis of the number of plagiarized sources correctly detected by each tool. It would also be possible to weight the different types of plagiarism because, in real teaching environments, the detection of single source plagiarism is usually less challenging than the case in which several sources have been mixed.

We will also improve the generational mechanism, so that it can code bigger and more complex submissions, not just toy problems: for instance, submissions with several functions or source files. This could be achieved by using smarter genetic operators and/or other different automatic programming techniques (e.g. classic GP trees [12]).

Finally, we think it is worth dedicating some effort to further studying the role of trash code in real teaching plagiarism identification.

The APL2 program used to generate the benchmarks and the four benchmarks themselves can be found at

http://manuelcebrianramos.googlepages.com/software

References

[1] ACM International Collegiate Programming Contest. [Online] Available: http://icpc.baylor.edu/icpc/.

[2] A. Aiken et al. Moss: A system for detecting software plagiarism. *University of California–Berkeley. See www. cs. berkeley. edu/aiken/moss. html*, 2005.

[3] M. Alfonseca and D. Selby. APL2 and PS/2: the Language, the Systems, the Peripherals. *APL Quote Quad (ACM SIGAPL)*, 19(4):1–5, Aug. 1989.

[4] B. Braumoeller and B. Gaines. Actions Do Speak Louder than Words: Deterring Plagiarism with the Use of Plagiarism-Detection Software. *PS: Political Science and Politics*, 34(04):835–839, 2002.

[5] G. Chaitin. *Algorithmic information theory*. Cambridge University Press New York, 1987.

[6] X. Chen, B. Francia, M. Li, B. McKinnon, and A. Seker. Shared information and program plagiarism detection. *Information Theory, IEEE Transactions on*, 50(7):1545–1551, 2004.

[7] R. Cilibrasi and P. Vitani. Clustering by Compression. *Information Theory, IEEE Transactions on*, 51(4):1523–1545, 2005.

[8] M. de la Cruz, A. Ortega, and M. Alfonseca. Attribute grammar evolution. *Lecture notes in computer science 3562*, pages 182–191.

[9] M. Freire, M. Cebrián, and E. del Rosal. *AC: An Integrated Source Code Plagiarism Detection Environment*. [Online] Available: http://tangow.ii.uam.es/ac/.

[10] D. Gitchell and N. Tran. Sim: a utility for detecting similarity in computer programs. *Technical Symposium on Computer Science Education*, pages 266–270, 1999.

[11] D. Grune and M. Vakgroep. Detecting copied submissions in computer science workshops. *Informatica Faculteit Wiskunde & Informatica, Vrije Universiteit*, 1989.

[12] J. Koza. *Genetic Programming: on the programming of computers by means of natural selection*. Bradford Book, 1992.

[13] T. Lancaster and F. Culwin. Towards an error free plagarism detection process. In *ITiCSE '01: Proceedings of the 6th annual conference on Innovation and technology in computer science education*, pages 57–60, New York, NY, USA, 2001. ACM Press.

[14] M. Li and P. Vitanyi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer, 1997.

[15] C. Lyon, R. Barrett, and J. Malcolm. Plagiarism is easy, but also easy to detect. *Plagiary*, 1, 2006.

[16] M. Marcotty, H. Ledgard, and G. Bochmann. A Sampler of Formal Definitions. *Computing Survey*, 8(2):181–276.

[17] M. O'Neill and C. Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*. Kluwer Academic Publishers, 2003.

[18] A. Ortega, M. de la Cruz, and M. Alfonseca. Christiansen Grammar Evolution: grammatical evolution with semantics. *IEEE Transactions on Evolutionary Computation*, Vol. 7:1, p.77-90, Jan. 2007.

[19] L. Prechelt, G. Malpohl, and M. Philippsen. Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science*, 8(11):1016–1038, 2002.

[20] J. Rissanen. Modeling by shortest data description. *Automatica*, 14(5):465–471, 1978.

[21] R. Solomonoff. The discovery of algorithmic probability. *Journal of Computer and System Sciences*, 55(1):73–88, 1997.

[22] G. Whale. Identification of Program Similarity in Large Populations. *The Computer Journal*, 33(2):140, 1990.

[23] M. Wise. Detection of similarities in student programs: YAP'ing may be preferable to plague'ing. *ACM SIGCSE Bulletin*, 24(1):268–271, 1992.